Chapter 11
Software testing tools

The use of testing tools can make testing easier, more effective and more productive. It is no coincidence that one of the first stated goals of attendees at testing courses is: What tools should we buy for my organization?

A wide variety of computer-aided software testing (CAST) tools are available, addressing many aspects of the testing process. Their scope and quality vary widely, and they provide varying degrees of assistance.

If we are to benefit from one of the major sources of leverage in the testing effort, a strategy for evaluation, acquisition, training, implementation, and maintenance is essential. It is an area where independent expertise can be enormously beneficial.

Categorizing test tools

There are a number of ways to categorize testing tools:

(1) *by the testing activity or task in which it is employed* – activities in this case include code verification, test planning, test execution;
(2) *by descriptive functional keyword* – the specific function performed by the tool such as capture/playback, logic coverage, comparator;
(3) *by major areas of classification* – a small number of high-level classes or groupings of tools.

Each class contains tools that are similar in function or other characteristics. Examples of these are test management tools, static analysis tools, and simulators.

The present chapter will focus on the activities with which the tool is associated, namely:

- reviews and inspections
- test planning

- test design and development
- test execution and evaluation
- test support.

We have chosen this approach of categorization for two reasons. First, it is closest to the point of view of the tester – what the tester is doing and when. Second, it is consistent with and based on the testing standards.

A listing of specific tools by descriptive function is included in Appendix G. In general, at the front end of the testing process there are fewer specialized tools available than at the back end.

Tools for reviews and inspections

Tools for reviews and inspections are the tools that assist with performing reviews, walkthroughs, and inspections of requirements, functional design, internal design, and code. Some tools are designed to work with specifications but there are far more tools available that work exclusively with code.

The types of tools required are:

- complexity analysis
- code comprehension
- syntax and semantic analysis.

Complexity analysis

Experienced programmers know that 20% of the code will cause 80% of the problems, and complexity analysis helps to find that all-important 20%. The McCabe Complexity Metrics were originally published in 1982 in the NBS (National Bureau of Standards) publication, "Structured Testing: A Software Testing Methodology Using the Cyclomatic Complexity Metric."

Complexity metrics identify high risk, complex areas. The cyclomatic complexity metric is based on the number of decisions in a program. It is important to testers because it provides an indication of the amount of testing (including inspections) necessary to practically avoid defects. In other words, areas of code identified as more complex are candidates for inspections and additional tests. There are other types of complexity metrics (e.g., from McCabe and Halstead), which in general are an indicator of program testing cost/schedule and number of defects in the program.

Code comprehension

Code comprehension tools help us understand unfamiliar code. They help us to understand dependencies, trace program logic, view graphical representations of the program, and identify dead code. They can be successfully used to identify areas that should receive special attention, such as areas to inspect.

There is a considerable amount of time spent in preparing for a code

inspections meeting. This requires extensive analysis, comprehension, and reverse engineering, all of which are made easier by code comprehension tools.

Syntax and semantic analysis

Syntax and semantic analysis tools perform extensive error checking to find errors that a compiler would miss, and are sometimes used to flag potential defects before or sometimes during formal testing.

These tools are language (and sometimes dialect) dependent. With the programming language C, for example, since there are a variety of dialects, these tools can often be configured by dialect. They parse code, maintain a list of errors, and provide build information. The parser can find semantic errors as well as make an inference as to what is syntactically incorrect.

Tools for test planning

The purpose of test planning is to define the scope, approach, resources (including tools), and schedule of testing activities. The test plan provides the foundation for the entire testing process, and, if this sounds like a cerebral activity, it should. Tools don't eliminate the need to think. As useful as capture/playback tools are (see the section on Test execution and evaluation below), they do not replace the need for sound test planning and design.

Perhaps the biggest help here comes from standards. The IEEE/ANSI Standard for Software Test Documentation (Std 829-1983) describes the purpose, outline, and content of the test plan, and the appendix of the standard includes examples taken from commercial data processing. Although a few tools have incorporated templates for test plans into them, many companies have found it useful to simply have someone enter the outline for test plans found in the IEEE/ANSI standard into an accessible edit file.

There are useful commercial tools that determine actual project staffing and schedule needs for adequate product testing. Often people rightly complain that schedules are predetermined by upper management, and the use of such a tool can provide an objective view of the realities of the project.

The types of tools required for test planning are:

- templates for test plan documentation
- test schedule and staffing estimates
- complexity analyzer.

Tools that help with reviews and inspections will also help with test planning, i.e., tools that identify complex product areas can also be used to locate areas that should impact planning for additional tests based on basic risk management.

Tools for test design and development

Test design is the process of detailing the overall test approach specified in the test plan for software features or combinations of features, and identifying and prioritizing the associated test cases. Test development is the process of translating the test design into specific test cases.

Like test planning, there's not a lot of help from test tools for the important, mostly mental process of test design. However, tools from the test execution and evaluation category, for example, capture/playback tools, assist with the development of tests, and are most useful as a means of implementing test cases that have been properly planned and designed.

The types of tools required for test design and development are:

- test data generator
- requirements-based test design tool
- capture/playback
- coverage analysis.

Once again, standards help. The IEEE/ANSI Standard for Software Test Documentation (Std 829-1983) describes the purpose, outline, and content of the test design specification, and the appendix of the standard also includes examples taken from commercial data processing.

Although a few tools have incorporated templates for the test design specification into them, many companies have found it useful to simply have someone enter the outline for test design specifications found in the IEEE/ANSI standard into an accessible edit file.

A useful type of tool in this category is a test data generation tool, which automates the generation of test data based on a user-defined format, for example, automatically generating all permutations of a specific, user-specified input transaction.

A tool that has not yet achieved widespread practical use is a requirements-based test design tool. Based on the assumption that faulty requirements can account for over 80% of the cost of errors, this highly disciplined approach based on cause-effect graph theory is used to design test cases to ensure that the implemented system meets the formally specified requirements document. This approach is for those who desire a disciplined, methodical, rigorous approach.

Test execution and evaluation tools

Test execution and evaluation is the process of executing test cases and evaluating the results. This includes selecting test cases for execution, setting up the environment, running the selected tests, recording the execution activities,

analyzing potential product failures, and measuring the effectiveness of the effort. Tools in the evaluation category assist with the process of executing test cases and evaluating the results.

The types of tools required for test execution and evaluation are:

- capture/playback
- coverage analysis
- memory testing
- simulators and performance.

Capture/playback

There is perhaps no chore more boring to the experienced tester than having to repeatedly re-run manual tests. Testers turn to capture/playback tools to automate the execution of tests, in other words, to run tests unattended for hours, overnight, or 24 hours a day if desired.

Capture/playback tools capture user operations including keystrokes, mouse activity, and display output. These captured tests, including the output that has been validated by the tester, form a baseline for future testing of product changes. The tool can then automatically play back the previously captured tests whenever needed and validate the results by comparing them to the previously saved baseline. This frees the tester from having to manually re-run tests over and over again when defect fixes and enhancements change the product.

Capture/playback tools can be classified as either native or non-intrusive. The native (sometimes called intrusive) form of capture/playback is performed within a single system. Both the capture/playback tool and the software being tested reside in the same system, i.e., the test tool is "native" to the system under test. It is sometimes called intrusive because the capture/playback software is distorting the operating performance to some extent, though for most software testing, this distortion is irrelevant.

The non-intrusive form of capture/playback requires an additional hardware system for the test tool. Usually the host system (containing the software under test) has a special hardware connection to the capture/playback tool, and this enables the capture/playback system to perform its required functions transparently to the host software. The best non-intrusive tools are platform and operating system independent.

There are three forms of capture/playback, listed in order of least to most expensive:

(1) *native/software intrusive* (introduces distortion at software level within the system under test);
(2) *native/hardware intrusive* (introduces distortion at hardware level only);
(3) *non-intrusive* (no distortion).

The most common type in use is native/software intrusive. Non-intrusive is typically used when the product being tested is itself an integrated hardware and software system where the introduction of additional internal hardware or software cannot be tolerated, e.g., real-time embedded systems. Since most software testing does not have this constraint, native/software intrusive is usually the cost-effective solution used by most organizations.

Choosing the right capture/playback tool turns out to be one of the most important and also most complex decisions an organization must make regarding testing. Unfortunately, capture/playback tool buyers are often forced to worry about such things as GUI test synchronization, proprietary testing languages, variable execution speed control, portability, multitasking testing, client/server, non-compare filtering, and non-intrusive testing. The best tools combine functionality with ease of use and mask many of the complexities of the tool's internal operation.

Coverage analysis

Coverage analyzers provide a quantitative measure of the quality of tests. In other words, they are a way to find out if the software is being thoroughly tested. This tool, essential to all software test organizations, tells us which parts of the product under test have in fact been executed (covered) by our current tests. They will tell us specifically what parts of the software product are not being covered, and therefore require more tests.

Some companies argue that it is not necessary to achieve full statement coverage, that is, to execute all of the statements within the product prior to release. They seem to think it is all right to expect customers to be the first to execute their code for them. Maybe these companies belong in another line of work. If we aren't measuring coverage, we do not have a handle on the job we are doing as testers.

Almost all structural tools run the source code into a preprocessor so that it can keep track of the coverage information. The problem is we now have a new source that's bigger than the old one so our object module is going to grow in size. The other possible problem is that performance may be impacted because we now have a different program than we did before. However, the final version of the software as delivered does not include the above preprocessing step, and therefore does not suffer this size and performance penalty.

There are many varieties of coverage, including statement, decision, condition, decision/condition, multiple condition, and path. As a minimum, the place to start is to make sure each statement in the program has been tested, and that each decision has taken on all possible outcomes at least once.

Memory testing

Whether being called bounds-checkers, memory testers, run-time error detectors, or leak detectors, in general the tools in this category include the ability to detect:

- memory problems
- overwriting and/or overreading array bounds
- memory allocated but not freed
- reading and using uninitialized memory.

Errors can be identified before they become evident in production and can cause serious problems. Detailed diagnostic messages are provided to allow errors to be tracked and eliminated.

Although memory testing tools tend to be language and platform specific, there are several vendors producing tools for the most popular environments. The top tools in this category are non-intrusive, easy to use, and reasonably priced. We put them in the "Just Do It" category, especially considering the alternative, i.e., shipping low-quality applications.

Test case management

The need for a test case management tool can creep up on us. We begin using a capture/playback tool to build and automate our tests. Then one day we wake up and find we have thousands of disorganized tests that need to be managed.

The best test case managers:

- provide a user interface for managing tests;
- organize tests for ease of use and maintenance;
- start and manage test execution sessions that run user-selected tests;
- provide seamless integration with capture/playback and coverage analysis tools;
- provide automated test reporting and documentation.

Why is test case management in a separate category, i.e., why isn't this incorporated into existing capture/playback tools? The bad news is – we're not there yet. The good news is that several of the leading tool vendors claim that they are working hard to accomplish this, and within the next year or so we should see several viable offerings.

Simulators and performance

Simulators take the place of software or hardware that interacts with the software to be tested. Sometimes they are the only practical method available for certain tests; for instance, when software interfaces with uncontrollable or unavailable hardware devices. They are frequently used to test telecommunications application programs, communications access methods, control programs, and networks.

Simulators also allow us to examine system performance. In general, performance tools help to determine what the software and system performance capabilities are. In practice, it is sometimes hard to find the line that distinguishes a simulator from a performance tool.

Finally, there are tools available for automated multi-user client/server load testing and performance measurement. These tools make it possible to create, control, and analyze the performance testing of client/server applications – before these applications go on line.

Software testing support tools

The test support category includes tools that, while not at the heart of the test process, lend overall support to the overall test effort. When these tools are of poor quality or do not exist, the professional tester suffers.

The types of tools required for test support are:

- problem management
- configuration management.

Problem management

Problem management tools are sometimes called defect tracking tools, bug management tools, incident control systems, etc., and are used to record, track, and generally assist with the management of defects and enhancements throughout the life cycle of software products.

Although many companies spend large sums developing home-grown problem management systems, there are tool vendors who now specialize in creating such systems across a variety of platforms. The best problem management tools are easy to customize for particular environments, and offer as standard features the capability to easily:

- and quickly submit and update defect reports;
- generate pre-defined or user-defined management reports;
- selectively notify users automatically of changes in defect status;
- provide secured access to all data via user-defined queries.

Configuration management

Configuration management (CM) is the key to managing, controlling, and coordinating changes to documents, and whatever else we really care about. CM tools assist the version control and build management process (see Chapter 6).

Besides problem management and configuration management, there are many tools not related to testing that in one way or another support the test process. These include project management tools, data base management software, spreadsheet software, and word processors.

Tool acquisition

The issues to be addressed in the acquisition of tools are largely good management common sense, but they are notoriously hard to implement. Far too often tools are purchased on an *ad hoc*, solve-the-short-term-crisis basis, and as a consequence end up on the shelf gathering dust and regarded as one more expensive experiment.

Making decisions about the acquisition of tools should involve some form of cost/benefit analysis, however simple. Tool vendors are naturally eager to tell us what their tool will do, and how it will solve our particular problems. The question we have to ask is: "At what cost?"

The important thing about cost is to establish true cost – meaning total cost or even lifetime cost. This will be a guesstimate, but it's a lot better than nothing, and as usual the purchase or license price is only the beginning. Additional costs are incurred in selection, installation, operation, training, maintenance and support, and the general cost of reorganizing procedures.

Tools that support testing processes which are already in place can be implemented with much less pain, human and financial, than those which will require an organizational clean sweep.

The scope of initial use needs to be established. Should the new tool be started with a single group, or even selected individuals, or should it be implemented organization-wide immediately?

Management support at the early stages is critical to ensure that the people who are actually working with the new tool can focus their attention on getting up and running and being productive as early as possible.

The careful choice and implementation of tools that support each other and have a synergistic effect is another major source of productivity.

The difficulty is finding the right tools from the right vendors. It is a major undertaking to produce and maintain a current evaluation of available testing tools and their capabilities. There are several hundred testing tools on the market, from companies that vary widely in size, installed customer base, product maturity, management depth, and understanding of testing and tools.

For more detail on tools and the tool vendor selection process, see Appendix G.

Independent expert advice at the early stages, preferably from the stage of evaluating processes and deciding what kinds of tools would be useful right up to the implementation stage, is invaluable. There is no doubt that testing can be easier, more effective, and more productive by using tools. By taking the proper steps, we can prevent an expensive new tool from becoming shelfware.

Questions before tool acquisition

There are a few questions that we recommend you answer as part of implementing an effective tools program:

- How do the tools fit into and support our test process?
- Do we know how to plan and design tests? (Tools do not eliminate your need to think, to plan, to design.)
- Who will be responsible for making sure we get the proper training on our new tool?
- Who will promote and support tool use within the organization on an ongoing basis?

Reference

IEEE/ANSI (1983). IEEE Standard for Software Test Documentation, (Reaff. 1991), IEEE Std 829-1983.

Chapter 12
Measurement

There are a number of big questions for testing – questions about product quality, risk management, release criteria, the effectiveness of the testing process, and when to stop testing.

Measurement provides answers. But once we start to think about what can be measured, it's easy to be overwhelmed with the fact that we could measure almost anything. However, this isn't practical (for the same reason that we can't test everything), and we have to create priorities for measurement based on what measures are critical and will actually be used once we have them.

> "Not everything that counts can be counted, and not everything that can be counted counts."
>
> Albert Einstein

Measurement for the sake of measurement can result in wasted effort. We should ask: "Is it useful? How will we profit from this measure? Is there an important question we could answer satisfactorily if we had reliable measurements in a particular area?" Or perhaps there is an important question we can't ask until we have a particular measurement.

Measurement provides answers

If our planning and subsequent activities are to be effective, they must be developed on a reliable, factual basis. How long should the testing take? How efficient is our testing? How good is our test library? Is it worth doing verification as opposed to validation? How thorough is our validation testing?

Based on past experience, what sort of state is the product likely to be in when it is declared "ready for testing?" What kinds of errors are we likely to find, how many, and where? How many errors probably remain in the product after testing? How is the product doing in test and production compared to other products on the market?

Measuring the number and types of errors detected during verification provides a measure of the efficiency of verification. Verification is expensive, and while we may be convinced that it is cost effective, this effectiveness will often need to be justified to others. How many errors are we picking up in validation that could have been found in verification?

Measuring validation test coverage (requirements, function, and logic coverage) provides quantitative assessments of the thoroughness and comprehensiveness of the validation tests. How much of the product are we testing? How good is our test library?

Measuring/tracking test execution status shows the convergence of key test categories (planned, available, executed, passed) and provides quantitative information on when to stop testing. How many of our tests are planned? How many are available? How many have been executed? How many have passed? When can we (reasonably) stop?

Program complexity provides answers useful when planning the size of the test effort and estimating the number of errors before testing begins.

Measuring and tracking of incident reports (by severity category) is a leading indicator of product quality. It provides an objective criteria for release readiness, a predictor of the number of remaining errors, and generally correlates to users' satisfaction with the product. When normalized, it provides a measure of product quality relative to other products.

When incident reports are not tracked, companies lose the handle on being able to fix problems responsibly. The backlog of incidents grows until it is so large that they no longer have a plan for managing it or getting it down to a reasonable size. Measurement provides an early warning that this situation is developing.

Customer-reported versus testing-reported incidents (only those diagnosed as confirmed errors) provide another measure of testing effectiveness. It is also important that testers have access to defects found by customers because these are a valuable basis for improving the test library (see Hetzel, 1993).

Useful measures

Measuring complexity

Generally, the more complex components of a program are likely to have more errors, and the more focused the testing effort must be to find them. There are plenty of good tools available off the shelf in this area, and it isn't necessary to understand exactly how these tools work to use them efficiently.

There are several well-known measures of program complexity. Most complexity measures can only be calculated after the program is written. Such measures are useful for certain validation tasks and defect prediction.

A simple measure of complexity that is fundamentally a measure of size is lines of code (LOC) or number of source statements, and can be counted in several different ways. A given line might be blank, a comment, one or more executable statements and/or data declarations. Also, there is the problem of comparing the LOC of programs written in different source languages. The simplest way to normalize the LOC is to generate the assembly-language equivalent program and then count LOC. We must decide how to count LOC and then standardize it within the organization. It is less critical to spend an inordinate amount of time debating exactly how to count than it is to count everything exactly the same way using the same tool.

An alternative to counting lines of code is function points. Like lines of code, function points are also a measure of size, effort, and complexity. Unlike lines of code, function points are derived from the user's perspective as detailed in functional specifications, and will stay constant independent of programming language. From the testing process perspective, function points have been used to estimate test cases required per function point and in measuring defects per function point.

In practice, debating lines of code versus function points often becomes a heated, religious discussion. Those that love function points see lines of code as the old, inaccurate approach with many significant shortcomings. The complaints about function points are that they are too abstract, do not relate as closely to what software developers actually produce, require training to learn how to count, and involve a process that seems complex to use to the uninitiated.

Both lines of code and function points today still have a place in software production and are worth considering. In the long term, as function points or a derivative become easy to understand, count, and use, they are likely to eventually completely replace the need to measure lines of code.

Another measure of complexity is McCabe's complexity metric which is the number of decisions (+1) in a program. An N-way branch is equivalent to N–1 decisions. Complexity across subprograms is additive. The result is a complexity number, which if it is above a certain limit, indicates the need for special attention, such as inspections or additional validation testing. Many of the leading tools provide an automatic capability for calculating the McCabe complexity metric.

Halstead's metrics are used for calculating program length (not to be confused with lines of code). Program length can be predicted before the program is written, and the predicted and actual values compare very closely over a wide range of programs. There are also formulas for predicting the number of bugs, programming effort, and time.

Measuring verification efficiency

The efficiency of verification activities is an important measure because verification is expensive. Verification test reports should contain lists of specific errors detected by each verification activity (see Chapter 10). Using subsequent error data from validation activities, one can count the errors detected and the errors missed (not detected) by verification.

Measuring test coverage

Coverage for requirements-based and function-based tests can be measured manually, using a requirements coverage/tracing matrix and a function coverage matrix. Logic coverage can be measured (practically) only with an automated tool. Measuring statement coverage is the most common practice in today's off-the-shelf coverage tools, and in general is a reasonable place to start.

Measuring/tracking test execution status

Test execution tracking is performed most simply by using a spreadsheet. The columns consist of a time stamp and the four test categories: planned, available, executed, passed. Each row is a periodic (e.g., weekly, daily) observation of the number of test cases in each category. The time between successive observations should be small enough to provide a sufficient number of observations to make any trends visible. The spreadsheet can be presented automatically in graphic form to make trends easier to interpret. A predetermined ratio of tests passed to tests planned (e.g., 98%) is often used as one criterion for release.

Measuring/tracking incident reports

Incident or "bug" reports can provide the basis of many valuable quality metrics for software. To realize the full potential of incident reports, the incident reporting process and its automated support system should follow a number of principles:

(1) There should be one and only one way to report an incident. Redundant reporting mechanisms create unnecessary work.

(2) There is a single, master repository for all incident reports. Fragmented, redundant, or incomplete data make it very difficult to obtain accurate, complete, and timely information.

(3) Every incident should be reported via the formal mechanism. Unreported incidents are never investigated. Informal reports often fall through the cracks, and their status cannot be tracked.

(4) Incident reports must rigorously identify the software configuration in which the incident occurred. Users must be able to dynamically obtain from the running system all necessary version information. Version identification can be trusted when it is obtained from the running system.

(5) Every user of a product, not just customers, should take the time to report incidents. This includes internal users, developers, testers, and administrators.

(6) After a new version of a product is baselined for formal testing and potential release (during the integration and test phase), all subsequent incidents should be formally reported. Problems detected before release are just as important as those detected after release.

(7) Every incident report should be investigated and then classified as one of the following:
   (i) user/operator error
   (ii) cannot reproduce the reported problem
   (iii) insufficient information
   (iv) documentation (user manual) error
   (v) request for change/enhancement
   (vi) confirmed product error
   (vii) other.

(8) Because a raw incident report describes the symptom of a problem (i.e., how the problem manifests itself to the user), the report must contain sufficient space for, or point to, the precise description of the real error. Many quality metrics are based on counts of confirmed errors, not incident reports. It must be possible to count the confirmed errors, not the symptoms of errors, in a specific product. This is not necessarily the product identified in the incident report. Multiple incident reports, each possibly describing different symptoms, should be able to point to the same underlying error without requiring a redundant description of the error in each incident report.

(9) For a confirmed problem, the report must also contain sufficient space for (or point to) the root cause categorization of the error. The best software engineering organizations have cultivated a culture within which a root cause analysis is performed for every confirmed error and then used in error prevention programs. Root causes are often rigorously categorized into a spectrum of standard causes.

(10) The incident reporting system should provide complete tracking information, from the time the report originates to the time it is formally closed. To manage the process, a state transition diagram should be developed to show all possible states of an incident report, the events which cause state changes, and the organizations authorized to make state changes.

(11) An incident report should not be considered closed until all work associated with the report is completed. In the case of a confirmed error, the correction should be available for customer use before the report is closed.

Test measures based on incident reports

The number of confirmed errors per 1,000 lines of code (errors/KLOC) is a common measure of error density that provides one indicator of product quality (or alternatively, the number of confirmed errors per function point). It is also an inter-product comparison of quality. The number of confirmed errors to date is itself a predictor of the number of remaining errors, because the number of undiscovered errors in a program is directly proportional to the number of errors already discovered.

A key component of release criteria usually states the number of confirmed, uncorrected errors (per severity level) that constitutes the limits of acceptability for release to customers. Comparing the number of confirmed errors reported by users and customers to the number reported by testing provides a measure of testing efficiency. The errors reported by users/customers also provide a basis for new test cases.

Other interesting measures

There are many measures of interest to software practitioners. A few of the common ones are:

(1) the age of a detected error;
(2) the response time to fix a reported problem;
(3) the percentage and frequency of detected errors by root-cause category;
(4) error removal efficiency;
(5) error cost:
   (i) cost of the failure to the user
   (ii) cost to investigate and diagnose
   (iii) cost to fix
   (iv) cost to retest
   (v) cost to release.

Recommendations

- Obtain consensus on the top three testing measurements to put in place in your organization.
- Put in place a capability for measuring/tracking test execution status based on the key test status model (planned, available, executed, passed).

- Locate a tool for measuring program complexity.
- Define the testing-related key release criteria and determine how to obtain the measurements.

References

Beizer, B. (1990). *Software Testing Techniques*. Van Nostrand Reinhold.
Capers Jones. (1991). *Applied Software Measurement*. McGraw-Hill.
Hetzel, W. (1993). *Making Software Measurement Work*. QED Publishing Group.
Humphrey, W.S. (1984). *Managing the Software Process*. Reading, MA: Addison-Wesley.
Kit, E. (1986a). *Testing C Compilers*, Computer Standards Conference.
Kit, E. (1986b). *State of the Art, C Compiler Testing*. Tandem Computers Technical Report.
IEEE/ANSI (1988a). IEEE Standard Dictionary of Measures to Produce Reliable Software, IEEE Std 982.1-1988.
IEEE/ANSI (1988b). IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software, IEEE Std 982.2-1988.

PART IV
Managing test technology

"Losing the test technology game is losing the software quality game which means losing the software game. Losing the software game is losing the computer game which is losing the high-tech game which in turn means losing the whole post-industrial society ball of wax."

BORIS BEIZER, Losing it, An Essay on World Class Competition

Chapter 13
Organizational approaches to testing

Most of us live our lives in organizations that change routinely. We live in structures that are never completely finished. Software testing in particular is an area where companies try many different approaches to organization – partly because of the perplexity about which structures are most effective, i.e., lead to most effective relations between people.

Why do organizations exist? What are the most fundamental, basic building blocks of structural design? What are the advantages and disadvantages of the specific approaches used in the real world to organize software testing? How can we decide which approach to use at this stage of our company?

Organizations exist because there is a need for people as a group to behave predictably and reliably. To achieve results, people need to cooperate with each other. Organizations are created to support, and more to the point, coerce the activities people engage in. Good organizations minimize the inevitable conflict between the needs of the organization and the needs of the individual.

For example, some software organizations have successfully created a positive culture that encourages and rewards migration of individuals from product development groups into independent test development groups equally as much as migration from test to development. In fact, the best cultures reward those who over time complete the full loop – start by spending a few years in development, then spend a few years in test, then return to development.

We have known a few exceptional people to repeat this entire loop twice over a span of about 10 years. The amazing thing is that these people are viewed as the most valued, the most prized resource by managers in both product development and test development. Talk about job security! Why? What better person to have in development than someone who knows how to design a test library? This developer will know how to create software that passes tests, because he or she knows how to think like a tester. And what better person to have designing tests than someone who knows how developers think?

There are many managers who unfortunately only see testing as a training ground for development, not a place to put their best developers. Too bad. I count myself among the managers who *will not hire* a person to do testing unless that person is qualified today to join the development group responsible for developing the same product that he or she would be testing. The good testing manager will encourage such a person to get a few years' experience in product development, and only then to consider moving to the testing group. Radical thinking? We don't think so!

Organizing and reorganizing testing

Once we have personally experienced several reorganizations, we learn to appreciate the need for organizational change to be very carefully planned. Let's face it – organizational change creates stress, disruption, demoralization, and confusion to varying degrees. The following 2200-year old quotation helps us to see how long the problems surrounding organizational change have been recognized and articulated:

> "We trained hard ... but it seemed that every time we were beginning to form up into teams we would be reorganized ... I was to learn later in life that we meet any new situation by reorganizing, and a wonderful method it can be for creating the illusion of progress while producing confusion, inefficiency, and demoralization."
>
> Petronius Arbiter, 210 BC

Isn't it frightening how relevant this thought from Petronius remains today?

An organization is a system that combines materials, knowledge, and methods in order to transform various kinds of inputs into valued outputs. Organizational structure is the responsibilities, authorities, and relationships, arranged in a pattern, through which the organization performs its functions. Structural choices include reporting hierarchies, job descriptions, and goals.

A test organization (usually called a test group) is a resource or set of resources dedicated to performing testing activities. As shops grow, the need for a dedicated, independent testing function becomes a more apparent necessity. It takes unbiased people to produce an unbiased measurement – testing must be done independently if it is to be fully effective in measuring software quality.

Organizational structures are like software. Neither are ever really completely finished. With time, the demands (external and internal) on the structure change – the capability of the existing structure to meet these demands decreases – making organizational redesign a repetitive activity. Likewise, with time, the requirements for a software system change. As with organizations, no matter how well it is designed at the beginning, eventually, if it is to remain useful, software must evolve. *We must leave behind the innocent delusion that once we understand the problem the software system is supposed to solve, we can go off alone to build and test it in peace.*

Test management is difficult. The manager of the test group must have the:

- ability to understand and evaluate software test process, standards, policies, tools, training, and measures;
- ability to maintain a test organization that is strong, independent, formal, and unbiased;
- ability to recruit and retain outstanding test professionals;
- ability to lead, communicate, support, and control;
- time to provide the care needed to manage test groups.

Senior managers must also have the ability to recruit and retain outstanding test managers. In fact, this is usually where the big mistakes are made. The senior managers do not understand the list above enough to know how to evaluate a potential test manager against these requirements. The result is that the wrong person is often promoted into test management!

When making organizational changes that affect testing, senior management needs to understand the impact the change will have on test management's ability to meet the above requirements. Plenty can go wrong when reorganizations occur without sufficient thought being given to the impact on testing. The dangers of having the wrong software testing structure include the following:

- Test independence, formality, and bias is weakened or eliminated.
- People in testing do not participate in reward programs.
- Testing becomes understaffed.
- Testing becomes improperly staffed with too many junior individuals.
- Testing is managed by far too junior managers.
- There is no leverage of test knowledge, training, tools, and process.
- Testing lacks the ability to stop the shipment of poor quality products.
- There is a lack of focused continuous improvement.
- Management lacks the bandwidth to manage testing groups.
- The quality focus is not emphasized.
- Test managers become demoralized owing to lack of career growth.

The above list can be used as a checklist of items to consider when planning organizational modifications. They can become questions; for example: Does making this change weaken the independence of testing? Does it hurt test's ability to get quality resources? etc.

Structural design elements

There is a surprisingly short list of basic building elements from which to construct a structure. The structural design elements are:

(1) *Tall or flat* – There may be many levels between the chief executive officer and the person on the shipping floor (this would be a tall organization), or there might be very few levels (a flat organization). In the last decade, flat has become more popular; managers finding themselves in the middle of a tall organization are particularly vulnerable to losing their jobs.
(2) *Market or product* – The organization may be structured to serve different markets or different products.
(3) *Centralized or decentralized* – The organization may be centralized or decentralized. This is a key question for the test organization. We will examine this in depth later in this chapter.
(4) *Hierarchical or diffused* – The organization may be hierarchical, that is, organized according to successively higher levels of authority and rank. Or it may be diffused, which is widely spread or scattered or matrixed.
(5) *Line or staff* – The organization will have a certain mix of line and/or staff roles.
(6) *Functional or project* – The organization may have functional or project orientations.

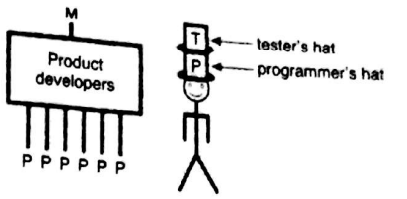
Combining these few design elements provides quite a variety of operating structural designs. Sometimes a variety of designs are implemented within the same company.

Approaches to organizing the test function

The above organizational basic design elements can be combined to produce many different test structures. There are seven approaches to organizing testing typically taken in practice that reflect the evolution of a maturing development organization. The following approaches to structure assume that unit testing should be done by product development. Therefore, the material that follows is about testing activities not related to unit testing, e.g., function testing, system testing, etc.

Approach 1. Testing is each person's responsibility

Approach 1 is what often occurs in the real world, especially when companies are small and little thought has been given to testing. As shown in Figure 13.1, there is a group of product developers whose primary responsibility is to



Advantages	Disadvantages
Seems on the surface like the most natural solution. (Evolves naturally out of the original product development team.)	Programmers are inherently incapable of effectively testing their own programs.

Figure 13.1 Approach 1: testing is each person's responsibility.

build a product. In this model, these product developers also are responsible for testing their own code. These people unfortunately must try their best to wear two hats, a programmer's hat and a tester's hat. They are responsible for function testing, system testing, and any other kind of testing that gets done.

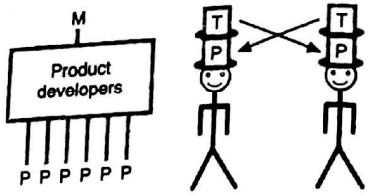
The problem with this approach is that it violates a basic assumption. Testing must be done independently if it is to be fully effective in measuring software quality. Programmers are biased by the creative work they do. This blinds them to their own errors; it is human nature.

Approach 2. Testing is each unit's responsibility

Approach 2 fixes the obvious flaw encountered in approach 1 – that of programmers being inherently incapable of testing their own programs – by assigning product developers within the group the job of testing each other's code. Note from Figure 13.2 that each person is still wearing two hats: the person on the right is responsible for product development of their own modules, plus they must test their team-mate's modules.

The problem now is for these people to find the time to understand the job of the software testing professional as well as understand product development processes, standards, policies, tools, training, metrics, etc. For typical software industry projects, this is just asking too much of one person. It is like expecting your average construction site hire to be a master electrician and a master carpenter on the same project at the same time. It is not impossible, it can occur – it is just not likely and does not make much sense.

In reality, these people will pick one hat to wear – the hat for which they know they have the primary responsibility and for which they are evaluated by management – the hat of the product developer. They will take time to test other people's code as time and skills permit. They will usually not get very good at learning the special skills, tools, and methods of testing while they are simultaneously responsible for developing product. That's reality.



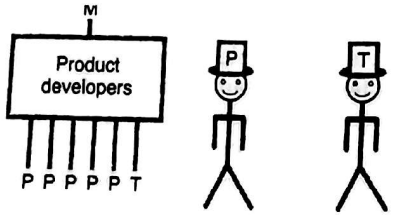
Advantages	Disadvantages
Solves the problem of programmers being incapable of testing their own code.	Programmer bandwidth (one hat too many).
	Methodologies, skills, training.

Figure 13.2 Approach 2: testing is each unit's responsibility.

Approach 3. Testing is performed by dedicated resource

Approach 3 solves the developer bandwidth issue by selecting a product developer and giving them a new job, that of test developer. Note from Figure 13.3 that each person in the group now only has to wear one professional hat. The tricky part here is for the group manager to pick the right person for testing.

In the evolutionary beginning, we noted that there was no test organization at all. One day the manager wakes up and declares, "All right, I see, I believe now. I know we need a dedicated person for testing. Hmmm, whom shall I tap to be our tester?" This really happens. Once while the author was presenting a course on testing, a development manager raised his hand and



Advantages	Disadvantages
Solves the developer bandwidth problem. (Only wearing one professional hat now).	Management bandwidth.
Single team.	Can management provide test process, standards, policies, tools, training, measures?

Figure 13.3 Approach 3: Testing performed by dedicated resource.

stated, "OK, OK, now I understand what testing is all about. I'm going to put our first full-time independent tester in place!" Great, thinks the course instructor, feeling very good about his teaching abilities.

"Yea", the manager continues, "I have this person that just doesn't fit on the development team. They've been a consistently poor performer; they're unhappy and they are just not contributing anything to reaching our project goals. I was going to fire them, but that is such an unpleasant task for me. I'll make them a tester! What have I got to lose?"

Of course we know he's got plenty to lose. The individual wearing too many hats now is the first line product development manager. This person, in addition to providing guidance to a product development team, must provide guidance on testing process, testing standards, testing policies, testing tools, testing training, testing measures, hiring professional testers, etc. It is just not going to happen.

It becomes clear that some sort of test group is necessary – a set of resources dedicated to performing testing activities and managed by a test manager. Without a formal organization, testing practices and tools must be set up for every project. With a separate test group, however, an organization remains in place to serve all projects on a continuing basis – to provide management with independent, unbiased, quality information.

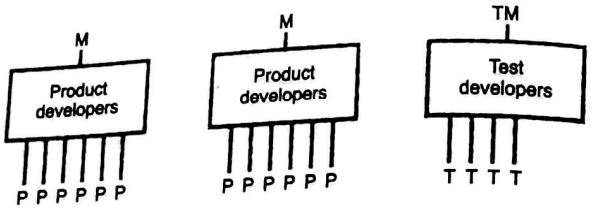
Bill Hetzel, in the book *The Complete Guide to Software Testing* (1988), tells us:

> An independent test organization is important because
> - building systems without one has not worked well,
> - effective measurement is essential to product quality control,
> - coordinating testing requires full-time, dedicated effort.

The creation of a formal test organization solves the problem of approach 3. Note in Figure 13.4 the importance of a test organization is realized, headed by a test manager. The question now becomes where to put the test group organizationally.

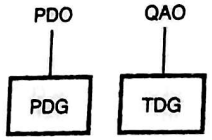
Approach 4. The test organization in QA

A common solution is to make the new test organization a component of quality assurance, where QA also audits the development process (see Figure 13.5). The manager of QA may not understand software testing. The capabilities of management of the testing group are critical, as is the manager of the testing manager. Since testing is performed in a separate organization from development, it will take extra effort to encourage and create a positive team environment. People also begin to ask, who owns quality? Development management complains that they no longer have the complete set of resources needed to produce a quality product.



Advantages	Disadvantages
Solves the management problems of approach 3.	Where to put it organizationally?

Figure 13.4 The test organization.



PDO = Product Development Organization
PDG = Product Development Group
QAO = Quality Assurance Organization
TDG = Test Development Group

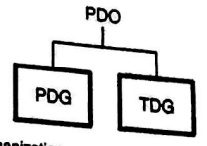
Advantages	Disadvantages
Solves the management problems of approach 3 by creating a formal test organization which is a part of QA.	Possible teamwork problems. TDG can get lost in QA group with other responsibilities. The product development group is not wholly accountable for producing quality products.

Figure 13.5 Approach 4: the test organization in QA.

Approach 5. The test organization in development

Approach 5 attempts to deal with the teamwork and quality ownership issue identified in approach 4 by creating a test organization that is part of the development organization (see Figure 13.6). This approach usually puts the test group under the second line product development manager.

Unfortunately, this is asking too much of most second line product development managers. It is much less an issue for higher level managers such as vice presidents that understand the need for a strong, independent



PDO = Product Development Organization
PDG = Product Development Group
TDG = Test Development Group

Advantages	Disadvantages
Solves the management problems of approach 3 by creating a formal test organization which is a part of development. Possibly solves teamwork issues of approach 4.	Does the senior development manager meet the test management requirements?

Figure 13.6 Approach 5: the test organization in development.

test function and who are willing to put a strong manager in place to manage the test group. In any event, all is riding on the senior manager who is now the one wearing two hats, that of managing the management responsible for product development and of managing the management responsible for software testing.

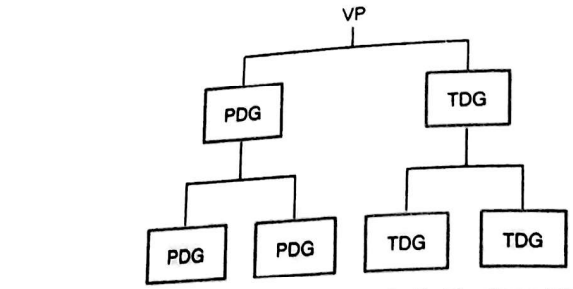
As organizations grow, more people are hired into the test organization, and multiple test groups are needed. A new issue arises. Should the multiple test groups be centrally or decentrally organized?

Approach 6. Centralized test organization

Approach 6 solves the senior management problem of approach 5 by creating a central test organization that lives within and serves a product development division (see Figure 13.7). Note how this creates a major opportunity for a senior test manager/director to significantly impact the organization. For example, the senior test manager can:

- manage the sharing of testing resources (both people and equipment) to smooth needs and better manage the risks of the company;
- coordinate consistent training for all testers across several test groups;
- promote consistent and high-quality testing tools for use by all testers;
- find and hire strong first line testing managers;
- provide real testing guidance to first line test managers.

In addition, first line testing managers see a potential career path in testing as they aspire to become a senior test manager.



Advantages	Disadvantages
Solves the senior management problems of approach 5 by centralizing the formal test organization which is a part of development.	Test organization lives or dies by VP.
Creates a career path for test managers.	Potential lack of teamwork at individual/project level.
	Possible lack of consistent test methodologies across VP organizations.

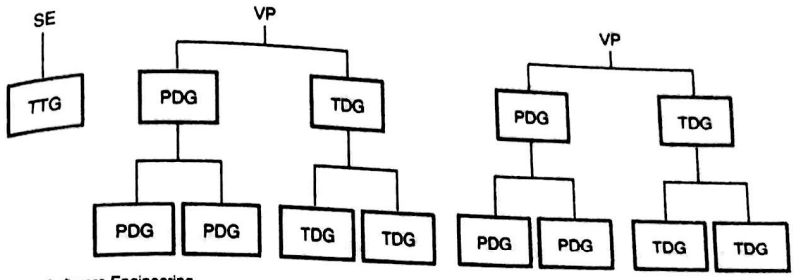
Figure 13.7 Approach 6: centralized test organization.

Now the risks are centralized, and assuming a good senior test manager is in place, the testing organization will live or die by the vice president who manages the product development and the test development organizations. When it's time to determine headcount, capital, and rewards for software testing, the vice president must provide the crucial support.

With the right vice president, this can and has often worked quite well in practice. A vice president that provides the proper resources, hires well, and delegates well to a highly competent senior test manager can make this approach work extremely well.

By creating project teams that are loaned resources from the independent test organization and that otherwise act, communicate, and live close together as a team alongside the product developers, the previously stated teamwork disadvantages are minimized. And when serious discussions regarding product quality are needed, a senior test manager is there to provide an unbiased voice to the vice president. This can be of great value to a vice president who cannot get the complete set of facts from product development management.

As companies continue to grow larger and contain multiple divisions, companies using this approach may begin to discover a lack of consistent approaches and best practices across different vice president organizations. This is solved by the next and final approach.



Advantages	Disadvantages
Solves the consistency problems of approach 6 by centralizing a test technology group which is a part of software engineering.	Test organizations live or die by VP.
	Potential lack of teamwork at individual/project level.

Figure 13.8 Approach 7: centralized and test technology.

Approach 7. Centralized test organization with a test technology center

Approach 7 solves the consistency problems of approach 6 by creating a test technology group, which in this case is part of a software engineering function (see Figure 13.8). This new test technology group is responsible for:

- leading and managing testing process and testing productivity improvement efforts;
- driving and coordinating testing training programs;
- coordinating the planning and implementation of testing tool programs;
- documenting test process, standards, policies, and guidelines as necessary;
- recognizing and leveraging best practices within the testing groups;
- recommending, obtaining consensus for, and implementing key testing measurements.

Selecting the right approach

How can we decide from among the many different approaches to using these structural design elements? To begin with, below is a set of sample selection criteria that can be used as a basis for evaluating different approaches. To what extent does the organizational structure:

- provide the ability for rapid decision making;
- enhance teamwork, especially between product development and testing development;
- provide for an independent, formal, unbiased, strong, properly staffed and rewarded, test organization;
- help to coordinate the balance of testing and quality responsibilities;
- assist with test management requirements as stated earlier in this chapter;
- provide ownership for test technology;
- leverage the capabilities of available resources, particularly people;
- positively impact morale and career path of employees (including managers)?

Providing for rapid decision making leads to improved responsiveness. It is possible to overcome an organizational structure that does not inherently provide for rapid decision making by creating special core teams and/or defining a special rapid escalation process, i.e., a daily high-level management short decision meeting used to escalate and deal with project issues.

One approach to doing reorganizations is to follow these steps:

(1) map the current organization;
(2) define and prioritize the key selection criteria (see list above);
(3) document new potential alternative approaches;
(4) evaluate potential approaches using a selection grid (see below);
(5) decide on a new organization;
(6) implement the new organization.

A decision matrix selection grid is a quality tool that can be used to evaluate the various approaches while using the selection criteria. Once the selection criteria are decided, they can be numbered or uniquely identified and placed across the top of the decision matrix.

The possible approaches are also numbered and placed vertically on the matrix as shown in Figure 13.9. Each approach is evaluated by considering each selection criteria and assigning a number to enter into the matrix. This number would come from a pre-defined range of values, e.g., from 1 to 5, where 5 indicates a high ranking, meaning the organization to a very great

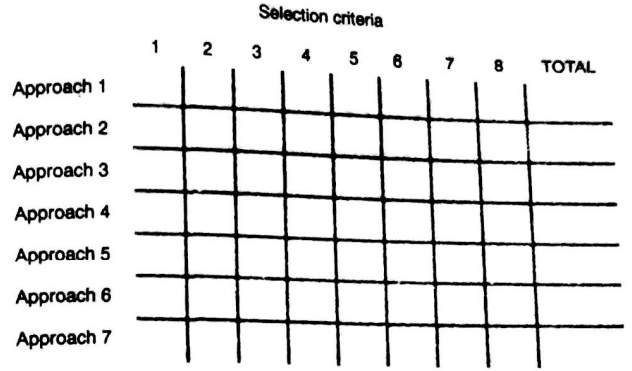


Figure 13.9 Decision matrix selection grid.

extent meets the given criteria for the approach under consideration. Likewise 3 might indicate a medium ranking and 1 a low ranking.

Add the scores horizontally to yield a total score for each approach. As a more advanced enhancement to this approach, the selection criteria can be assigned a multiplier weighting to be factored in.

Reorganizations should be carefully planned and implemented. One last piece of advice – remember to involve the participants in the above process!

References

Beizer, B. (1992). "Losing It, An Essay on World Class Competition," *Software Quality World*, 4(3).

Goodman, Paul, Sproull & Associates (1990). *Technology and Organizations*. San Francisco, CA: Jossey-Bass.

Gelperin, D. and Hetzel, W. (1989). *STEP: Introduction and Summary Guide*. Jacksonville, FL: Software Quality Engineering.

Hetzel, W. (1988). *The Complete Guide to Software Process*. Wellesley, MA: QED Information Sciences.

Humphrey, W.S. (1989). *Managing the Software Process*. Reading, MA: Addison-Wesley.

Kaner, C. (1988). *Testing Computer Software*. Blue Ridge Summit, PA: TAB Books.

Kit, E. (1992). "Approaches to Organizing the Independent Test Function," *Software Testing Analysis & Review (STAR) Conference Proceedings*.

Silverman, M. (1984). *The Technical Manager's Survival Book*. New York: McGraw-Hill.

Chapter 14

Current practices, trends, challenges

Advances in technology and development tools have enabled the creation of complex software using graphical user interfaces (GUIs) and client/server application architectures. Although these advances have improved development productivity, the resulting software is difficult to test, and the new demands on testing are eroding the development productivity gains achieved. Automation through commercial testing tools is stepping in to fill part of the gap.

During the short history of software engineering, there has been a dramatic shift in tester-to-developer ratios (in favor of the testers) which indicates that the importance of testing is now recognized, and organizations are now more willing to pay for proper testing. At the same time the number of testers continues to grow, and time spent on testing and its perceived effectiveness are increasing.

Meanwhile, studies of best practice in the industry indicate that the strictly non-technical basics – good management, communications, people, and controls – are more important to testing success than state-of-the-art methodologies, CASE tools, and other silver bullets.

GUIs: What's new here ?

There are two types of user interfaces:

(1) character-based user interfaces (traditional style);
(2) GUIs.

GUIs are characterized by the use of a different input device (mouse) and by high-resolution spatial and color images. GUI testing requires increased automation, but this is much more complicated in a GUI environment than a character-based environment. Fortunately, the new generation of capture/replay tools provides the answer (see Chapter 11).

Usage testing

Knowing how the product is really going to be used enables us to focus on certain types of critical areas when testing. When we prioritize validation tests, we are already, usually subconsciously, making judgments about potential usage. Usage testing is a more formal approach to this problem.

Usage testing is the process of:

- initial testing based on *estimated* usage patterns;
- measuring (collecting data on) the actual usage (after the product is complete and functional), and developing an *operational profile*;
- adjusting priorities, developing new tests, and retesting, based on the operational profile.

Usage testing reflects expected operational use in hardware and software configurations, in the frequency of operations tested, in the sequence of operations tested and in system load. Testing emphasis is on detecting errors that are the most likely to occur in operational use.

Usage testing is ideally performed after unit and integration testing, when substantial functionality is available, and after requirements validation – if it is performed separately. In practice it is usually performed after developing operational profiles based on real usage by real users, and this may mean that usage testing can apply only to releases after the first.

The advantages of usage testing are that it is customer oriented; test resources and schedules are optimized to maximize customer satisfaction. The hard part is cost-effectively obtaining operational profiles, which is why many companies today are not formally doing usage testing. However, most leading software companies do factor-in known or expected customer usage of the product when prioritizing tests worth developing.

Usage testing was pioneered by Frank Ackerman (1992, 1993), formerly of the Institute for Zero Defect Software.

Tester-to-developer ratios

Tester-to-developer staffing ratios are an interesting indication of the number of people performing testing activities compared to the number of people developing the software product. Ratios have improved dramatically in favor of more testers during the short and stormy life of the software engineering discipline.

Historically, for mainframes, the tester-to-developer ratio was 1:5–10, meaning one tester for every five to ten product developers. More recently published numbers include:

• Microsoft, 1992	2:3
• Lotus (for 1-2-3 Windows)	2:1
• Average of 4 major companies (1992)	1:2

(*Note*: The Microsoft and Lotus reference is Marvin (1993). The Microsoft ratio is for the Microsoft Worldwide Product Division. The four companies referenced are Microsoft, Borland, WordPerfect, and Novell for which the reference is Norman (1993).)

The above figures should be viewed as above average, that is, applying more testers than most companies do in practice. Informal surveys performed during testing courses taught by Software Development Technologies indicate that most companies do not enjoy ratios as good as one testing professional for every two product developers. More typical ratios are in the range from 1:3 to 1:4. Most testing professionals surveyed from companies operating at 1:5 or above (e.g., 1:7, 1:10, or even 1:20) generally felt that testing was under-resourced.

Software measures and practices benchmark study

This important study was a joint initiative by Xerox Corporation and Software Quality Engineering to improve understanding of software excellence. The goal of the study was to identify world-class projects in software technology and to characterize the engineering, management and measurement practices of these projects. A full research report is available from Software Quality Engineering (1991).

The design of the study included two phases. The purpose of phase 1 was to identify the "best" projects. Seventy-five software organizations were surveyed and responded to 52 questions (15 on the organization and 37 on its practices). The purpose of phase 2 was to characterize the practices of the "best" projects. Ten "best" projects (7 companies, 509 people) were surveyed, and responded to 104 questions (17 on attitude, 87 on practices and measures) and were subjected to an on-site practices assessment.

"Best" projects were selected on the following basis:

- perceived as producing high-quality results;
- perceived as using better practices;
- implemented recently or in final test with project team accessible for interview.

In other words, the best practices were reverse engineered from projects identified subjectively as having the highest quality – the projects the organization was the most proud of.

Companies were selected for phase 2 on the following basis:

- software a significant part of organizational mission;
- high scores on Phase 1 survey (above 75th percentile);
- reputation and public perception of success.

The actual phase 2 companies were:

- AT&T
- Dupont
- GTE
- IBM
- NCR
- Siemens
- Xerox.

The key findings of the study were:

(1) best projects emphasize strong up-front planning and close tracking and reporting of status on an ongoing basis;
(2) best projects rely on management fundamentals (teamwork, communication, and controls), not on technology and state-of-the-art methodologies;
(3) best projects emphasize reviews, inspections, and very strong and independent high-level testing;
(4) measurement used to track progress, quality problems, and issues.
(5) seven practices in common use related to planning and up-front requirements and design specifications;
(6) best projects utilized recommended practices – the investment required to achieve a more disciplined and structured process is evident in the results;
(7) best projects excel in different areas (survey categories) and emphasize different phases of the life cycle; no single project was superior in all areas.

The key measurement practices were discovered to be:

- schedule performance

- code defects
- test results
- test defects
- defects after release
- number of open problems
- time to correct problems
- issue tracking
- lines of code
- process compliance.

There are no silver bullets here. The keys to success are basic issues:

- good management
- good communications
- good people
- good controls
- ongoing measurements.

References

Ackerman, F. (1993). "Usage Testing," *Software Testing Analysis & Review (STAR) Conference Proceedings 1992.*

Ackerman, F. (1994). "Constructing and using Operational Profiles," *Software Testing Analysis & Review (STAR) Conference Proceedings 1993.*

Norman, S. (1993). "Testing GUIs is a sticky business," *Software Testing Analysis & Review (STAR) Conference Proceedings 1992.*

Software Quality Engineering (1991) Report 908.

Tener, M. (1993). "Testing in the GUI and Client/Server World," *IBM OS/2 Developer*, Winter.

Chapter 15
Getting sustainable gains in place

Only with a strong commitment to proven software engineering practices can companies successfully compete, survive, and thrive in today's marketplace. The necessary changes and investments to advance the state of the practice must be made; the alternative is to lose everything. Years of consulting experience, corroborated by the Xerox study (see Chapter 14), makes it clear that if we focus on planning, communication, tracking, and measurement, we'll substantially increase our odds of creating products with the right level of quality.

Getting gains to happen

Being an agent for change is hard. Whether you're striving for small or large changes, tactical or strategic improvements, those that require cultural evolution or revolution, take the time to assess the situation and plan for gains that are sustainable. Before proceeding, take a step back, look carefully around you. Are you facing small cracks in the pavement ahead or a deep crevasse? The strategy will be different for each. Taking the recurrent advice of making small steps in the name of continuous improvement when facing a large crevasse can be disastrous! Remember, as the Chinese proverb says:

> "It is very dangerous to try and leap a chasm in two bounds."

Changing an organization is hard. Change must be positive, planned, and controlled, with a minimum of disruption, and a minimum of culture shock. Try to determine the readiness for change: How much pain is the organization feeling now? How much pain are our customers feeling now? What is the internal level of dissatisfaction with the status-quo versus the level of natural resistance to change (Migdoll, 1993)?

The benefits of change have to be defined, and the expected improvements justified. Change has to be sold to managers and practitioners throughout the organization. Measure the effectiveness of changes that have been